2. Some benefits of the template approach are the guaranteed type-safety at compile-time as we can generate a concrete version of the function/class with the specified type. It also enables the same runtime efficiency as a custom version must be generated for every distinct type. Therefore, we can optimize the code as if it was written for a dedicated function for that type.

Some benefits of the generics approach include the built-in type safety due to the enforcement of equivalent operations and it allows you to write code with less explicit type casts.

Some drawbacks of the template approach include the increased compilation time due to the instance of the function/class needing to be generated for each type. Error messages might also be hard to implement as they could only apply to generic templated instance which may result in them being nested deep inside the templates.

Some drawbacks of the generics approach include that they are more conservative and thus are less flexible in what types of operations and functionality are allowed to be performed using generics.

Some benefits of the duck-typing approach is that it is very flexible and allows for polymorphism without inheritance. It can also result in more readable code.

Some drawbacks of the duck-typing approach include that errors are only caught at runtime which makes debugging more difficult and there might be some performance overhead due to typechecking at runtime.

3. Dynamically Typed cannot support parametric polymorphism because we don't can't specify types for variables in dynamic languages. Thus we can't actually have "type-parameterized" classes or functions. However, we are able to accomplish the same functionality with regular functions/classes.

4. We can utilize templates (in particular specialization) to accomplish duck-typing in statically typed languages. We can then define different behavior for different types. It is similar to the dynamically-typed version because we are able to have different behaviors for different types. We also are able to reuse the same functions/classes for different types. It is different because the type-checking will still be performed at compile time and thus errors will be caught earlier in the development process. Additionally, the types that a function/class must be known at compile-time.

6. Although Java-script doesn't necessrily have classes we can construct uniform sets of objects that all have the same methods/fields in the following ways:

We can create object literals which will represent prototypes and we can then utilize object create to have the created object inherit the prototype.

```
var prototypeObject = {
    method1: function() { ... },
    method2: function() { ... },
    field1: 'value1',
    field2: 'value2'
};

var newObject = Object.create(prototypeObject);
```

We can also use constructor functions using the new. to create a new object with the specified methods and fiedls and the this. keywords.

```
function MyObject() {
    this.field1 = 'value1';
    this.field2 = 'value2';
    this.method1 = function() { ... };
    this.method2 = function() { ... };
}

var newObject = new MyObject();
```

8. We cannot use the IShape type to instantiate x because IShape is an abstract base class due to its pure virtual function. And in C++, we are unable to create an instance of an abstract base class. The IShape *ptr line is able to work since we are creating a pointer to an IShape object and not an instance itself. Thus, we can use the pointer to point to instances of classes that inherit from IShape and implement all of the pure virtual functions.